Associative Long Short-Term Memory

Ivo Danihelka   DANIHELKA@GOOGLE.COM
Greg Wayne   GREGWAYNE@GOOGLE.COM
Benigno Uria   BURIA@GOOGLE.COM
Nal Kalchbrenner   NALK@GOOGLE.COM
Alex Graves   GRAVESA@GOOGLE.COM
Google DeepMind

Abstract

We investigate a new method to augment recurrent neural networks with extra memory without increasing the number of network parameters. The system has an associative memory based on complex-valued vectors and is closely related to Holographic Reduced Representations and Long Short-Term Memory networks. Holographic Reduced Representations have limited capacity: as they store more information, each retrieval becomes noisier due to interference. Our system in contrast creates redundant copies of stored information, which enables retrieval with reduced noise. Experiments demonstrate faster learning on multiple memorization tasks.

1. Introduction

We aim to enhance LSTM (Hochreiter & Schmidhuber, 1997), which in recent years has become widely used in sequence prediction, speech recognition and machine translation (Graves, 2013; Graves et al., 2013; Sutskever et al., 2014). We address two limitations of LSTM. The first limitation is that the number of memory cells is linked to the size of the recurrent weight matrices. An LSTM with N_h memory cells requires a recurrent weight matrix with $O(N_h^2)$ weights. The second limitation is that LSTM is a poor candidate for learning to represent common data structures like arrays because it lacks a mechanism to index its memory while writing and reading.

To overcome these limitations, recurrent neural networks have been previously augmented with soft or hard attention mechanisms to external memories (Graves et al., 2014; Sukhbaatar et al., 2015; Joulin & Mikolov, 2015; Grefen-

stette et al., 2015; Zaremba & Sutskever, 2015). The attention acts as an addressing system that selects memory locations. The content of the selected memory locations can be read or modified by the network.

We provide a different addressing mechanism in Associative LSTM, where, like LSTM, an item is stored in a distributed vector representation without locations. Our system implements an associative array that stores key-value pairs based on two contributions:

1. We combine LSTM with ideas from Holographic Reduced Representations (HRRs) (Plate, 2003) to enable key-value storage of data.

2. A direct application of the HRR idea leads to very lossy storage. We use redundant storage to increase memory capacity and to reduce noise in memory retrieval.

HRRs use a "binding" operator to implement key-value binding between two vectors (the key and its associated content). They natively implement associative arrays; as a byproduct, they can also easily implement stacks, queues, or lists. Since Holographic Reduced Representations may be unfamiliar to many readers, Section 2 provides a short introduction to them and to related vector-symbolic architectures (Kanerva, 2009).

In computing, Redundant Arrays of Inexpensive Disks (RAID) provided a means to build reliable storage from unreliable components. We similarly reduce retrieval error inside a holographic representation by using redundant storage, a construction described in Section 3. We then combine the redundant associative memory with LSTM in Section 5. The system can be equipped with a large memory without increasing the number of network parameters. Our experiments in Section 6 show the benefits of the memory system for learning speed and accuracy.

2. Background

Holographic Reduced Representations are a simple mechanism to represent an associative array of key-value pairs in a fixed-size vector. Each individual key-value pair is the same size as the entire associative array; the array is represented by the sum of the pairs. Concretely, consider a complex vector key $r = (a_r[1]e^{i\phi_r[1]}, a_r[2]e^{i\phi_r[2]}, \dots)$, which is the same size as the complex vector value x. The pair is "bound" together by element-wise complex multiplication, which multiplies the moduli and adds the phases of the elements:

$$y = r \circledast x \tag{1}$$
$$= (a_r[1]a_x[1]e^{i(\phi_r[1]+\phi_x[1])}, a_r[2]a_x[2]e^{i(\phi_r[2]+\phi_x[2])}, \dots) \tag{2}$$

Given keys r_1, r_2, r_3 and input vectors x_1, x_2, x_3, the associative array is

$$c = r_1 \circledast x_1 + r_2 \circledast x_2 + r_3 \circledast x_3 \tag{3}$$

where we call c a *memory trace*.

Define the *key inverse*:

$$r^{-1} = (a_r[1]^{-1}e^{-i\phi_r[1]}, a_r[2]^{-1}e^{-i\phi_r[2]}, \dots) \tag{4}$$

To retrieve the item associated with key r_k, we multiply the memory trace element-wise by the vector r_k^{-1}. For example:

$$r_2^{-1} \circledast c = r_2^{-1} \circledast (r_1 \circledast x_1 + r_2 \circledast x_2 + r_3 \circledast x_3)$$
$$= x_2 + r_2^{-1} \circledast (r_1 \circledast x_1 + r_3 \circledast x_3)$$
$$= x_2 + noise \tag{5}$$

The product is exactly x_2 together with a noise term. If the phases of the elements of the key vector are randomly distributed, the noise term has zero mean.

Instead of using the key inverse, Plate recommends using the complex conjugate of the key, $\overline{r_k} = (a_k[1]e^{-i\phi_k[1]}, a_k[2]e^{-i\phi_k[2]}, \dots)$, for retrieval: the elements of the exact inverse have moduli $(a_k[1]^{-1}, a_k[2]^{-1}, \dots)$, which can magnify the noise term. Plate presents two different variants of Holographic Reduced Representations. The first operates in real space and uses circular convolution for binding; the second operates in complex space and uses element-wise complex multiplication. The two are related by Fourier transformation. See Plate (2003) or Kanerva (2009) for a more comprehensive overview.

3. Redundant Associative Memory

As the number of items in the associated memory grows, the noise incurred during retrieval grows as well. Here,

we propose a way to reduce the noise by storing multiple transformed copies of each input vector. When retrieving an item, we compute the average of the restored copies. Unlike other memory architectures (Graves et al., 2014; Sukhbaatar et al., 2015), the memory does not have a discrete number of memory locations; instead, the memory has a discrete number of copies.

Formally, let c_s be the memory trace for the s-th copy:

$$c_s = \sum_{k=1}^{N_{items}} (P_s r_k) \circledast x_k \tag{6}$$

where $x_k \in \mathbb{C}^{N_h/2}$ is the k-th input; $r_k \in \mathbb{C}^{N_h/2}$ is the key. Each complex number contributes a real and imaginary part, so the input and key are represented with N_h real values. $P_s \in \mathbb{R}^{N_h/2 \times N_h/2}$ is a constant random permutation, specific to each copy. Permuting the key decorrelates the retrieval noise from each copy of the memory trace.

When retrieving the k-th item, we compute the average over all copies:

$$\tilde{x}_k = \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} \overline{(P_s r_k)} \circledast c_s \tag{7}$$

where $\overline{(P_s r_k)}$ is the complex conjugate of $P_s r_k$.

Let us examine how the redundant copies reduce retrieval noise by inspecting a retrieved item. If each complex number in r_k has modulus equal to 1, its complex conjugate acts as an inverse, and the retrieved item is:

$$\tilde{x}_k = x_k + \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} \sum_{k' \neq k}^{N_{items}} \overline{(P_s r_k)} \circledast (P_s r_{k'}) \circledast x_{k'}$$
$$= x_k + \sum_{k' \neq k}^{N_{items}} x_{k'} \circledast \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} P_s(\overline{r_k} \circledast r_{k'})$$
$$= x_k + noise \tag{8}$$

where the $\sum_{k' \neq k}^{N_{items}}$ sum is over all other stored items. If the terms $P_s(\overline{r_k} \circledast r_{k'})$ are independent, they add incoherently when summed over the copies. Furthermore, if the noise due to one item $\xi_{k'} = x_{k'} \circledast \frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} P_s(\overline{r_k} \circledast r_{k'})$ is independent of the noise due to another item, then the variance of the total noise is $O(\frac{N_{items}}{N_{copies}})$. Thus, we expect that the retrieval error will be roughly constant if the number of copies scales with the number of items.

Practically, we demonstrate that the redundant copies with random permutations are effective at reducing retrieval noise in Figure 1. We take a sequence of ImageNet images (Russakovsky et al., 2015), each of dimension $3 \times 110 \times 110$, where the first dimension is a colour channel. We vectorise each image and consider the first half of the vector
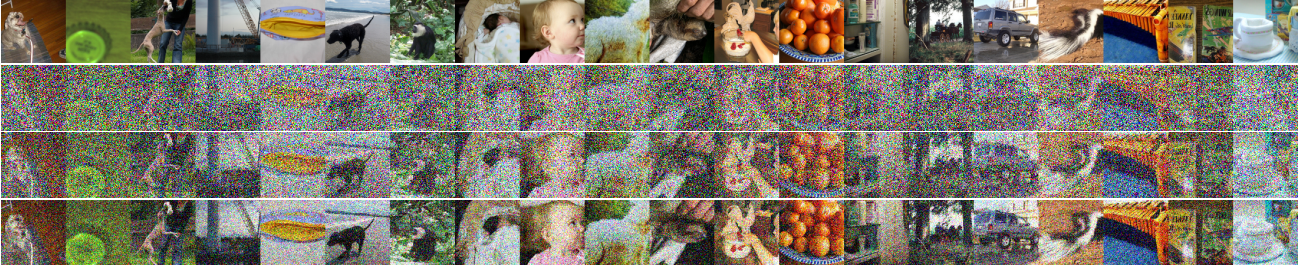
Figure 1. From top to bottom: 20 original images and the image sequence retrieved from 1 copy, 4 copies, and 20 copies of the memory trace. Using more copies reduces the noise.
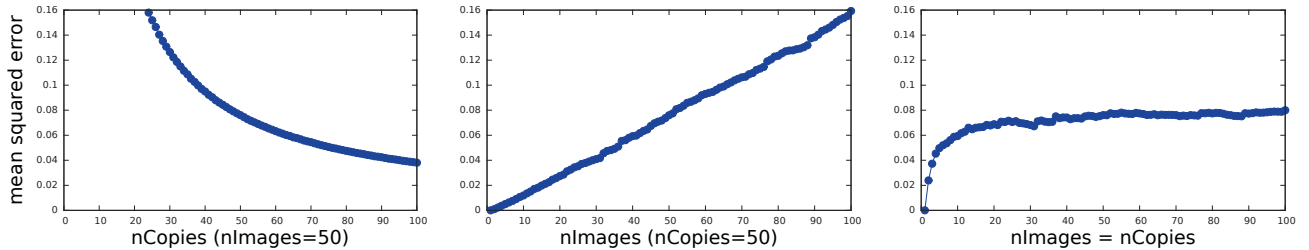

Figure 2. The mean squared error per pixel when retrieving an ImageNet image from a memory trace. **Left:** 50 images are stored in the memory trace. The number of copies ranges from 1 to 100. **Middle:** 50 copies are used, and the number of stored images goes from 1 to 100. The mean squared error grows linearly. **Right:** The number of copies is increased together with the number of stored images. After reaching 50 copies, the mean squared error is almost constant.

to be the real part and the second half the imaginary part of a complex vector. The sequence of images is encoded by using random keys with moduli equal to 1. We see that retrieval is less corrupted by noise as the number of copies grows.

The mean squared error of retrieved ImageNet images is analysed in Figure 2 with varying numbers of copies and images. The simulation agrees accurately with our prediction: the mean squared error is proportional to the number of stored items and inversely proportional to the number of copies.

The redundant associative memory has several nice properties:

- The number of copies can be modified at any time: reducing their number increases retrieval noise, while increasing the number of copies enlarges capacity.

- It is possible to query using a partially known key by setting some key elements to zero. Each copy of the permuted key $P_s r_k$ routes the zeroed key elements to different dimensions. We need to know $O(\frac{N_h}{N_{copies}})$ elements of the key to recover the whole value.

- Unlike Neural Turing Machines (Graves et al., 2014), it is not necessary to search for free locations when writing.

- It is possible to store more items than the number of copies at the cost of increased retrieval noise.

4. Long Short-Term Memory

We briefly introduce the LSTM with forget gates (Gers et al., 2000), a recurrent neural network whose hidden state is described by two parts $h_t, c_t \in \mathbb{R}^{N_h}$. At each time step, the network is presented with an input x_t and updates its state to

$$\hat{g}_f, \hat{g}_i, \hat{g}_o, \hat{u} = W_{xh} x_t + W_{hh} h_{t-1} + b_h \quad (9)$$
$$g_f = \sigma(\hat{g}_f) \quad (10)$$
$$g_i = \sigma(\hat{g}_i) \quad (11)$$
$$g_o = \sigma(\hat{g}_o) \quad (12)$$
$$u = \tanh(\hat{u}) \quad (13)$$
$$c_t = g_f \odot c_{t-1} + g_i \odot u \quad (14)$$
$$h_t = g_o \odot \tanh(c_t) \quad (15)$$

where $\sigma(x) \in (0, 1)$ is the logistic sigmoid function, and g_f, g_i, g_o are the *forget gate*, *input gate*, and *output gate*, respectively. The u vector is a proposed *update* to the *cell state* c. W_{xh}, W_{hh} are weight matrices, and b_h is a bias vector. "$\odot$" denotes element-wise multiplication of two vectors.

5. Associative Long Short-Term Memory

When we combine Holographic Reduced Representations with the LSTM, we need to implement complex vector multiplications. For a complex vector $z = h_{real} + ih_{imaginary}$, we use the form

$$h = \left[\begin{array}{c} h_{real} \\ h_{imaginary} \end{array} \right] \qquad (16)$$

where $h \in \mathbb{R}^{N_h}$, $h_{real}, h_{imaginary} \in \mathbb{R}^{N_h/2}$. In the network description below, the reader can assume that all vectors and matrices are strictly real-valued.

As in LSTM, we first compute gate variables, but we also produce parameters that will be used to define associative keys $\hat{r}_i, \hat{r}_o$. The same gates are applied to the real and imaginary parts:

$$\hat{g}_f, \hat{g}_i, \hat{g}_o, \hat{r}_i, \hat{r}_o = W_{xh}x_t + W_{hh}h_{t-1} + b_h \qquad (17)$$

$$\hat{u} = W_{xu}x_t + W_{hu}h_{t-1} + b_u \qquad (18)$$

$$g_f = \left[\begin{array}{c} \sigma(\hat{g}_f) \\ \sigma(\hat{g}_f) \end{array} \right] \qquad (19)$$

$$g_i = \left[\begin{array}{c} \sigma(\hat{g}_i) \\ \sigma(\hat{g}_i) \end{array} \right] \qquad (20)$$

$$g_o = \left[\begin{array}{c} \sigma(\hat{g}_o) \\ \sigma(\hat{g}_o) \end{array} \right] \qquad (21)$$

Unlike LSTM, we use an activation function that operates only on the modulus of a complex number. The following function restricts the modulus of a (*real*, *imaginary*) pair to be between 0 and 1:

$$\text{bound}(h) = \left[\begin{array}{c} h_{real} \oslash d \\ h_{imaginary} \oslash d \end{array} \right] \qquad (22)$$

where "$\oslash$" is element-wise division, and $d \in \mathbb{R}^{N_h/2} = \max(1, \sqrt{h_{real} \odot h_{real} + h_{imaginary} \odot h_{imaginary}})$ corresponds to element-wise normalisation by the modulus of each complex number when the modulus is greater than one. This hard bounding worked slightly better than applying $\tanh$ to the modulus. "bound" is then used to construct the *update* and two keys:

$$u = \text{bound}(\hat{u}) \qquad (23)$$

$$r_i = \text{bound}(\hat{r}_i) \qquad (24)$$

$$r_o = \text{bound}(\hat{r}_o) \qquad (25)$$

where $r_i \in \mathbb{R}^{N_h}$ is an *input key*, acting as a storage key in the associative array, and $r_o \in \mathbb{R}^{N_h}$ is an *output key*, corresponding to a lookup key. The update u is multiplied with the input gate g_i to produce the value to be stored.

Now, we introduce redundant storage and provide the procedure for memory retrieval. For each copy, indexed by $s \in \{1, \dots, N_{copies}\}$, we add the same key-value pair to the cell state:

$$r_{i,s} = \left[\begin{array}{cc} P_s & 0 \\ 0 & P_s \end{array} \right] r_i \qquad (26)$$

$$c_{s,t} = g_f \odot c_{s,t-1} + r_{i,s} \circledast (g_i \odot u) \qquad (27)$$

where $r_{i,s}$ is the permuted input key; $P_s \in \mathbb{R}^{N_h/2 \times N_h/2}$ is a constant random permutation matrix, specific to the s-th copy. "$\circledast$" is element-wise complex multiplication and is computed using

$$r \circledast u = \left[\begin{array}{c} r_{real} \odot u_{real} - r_{imaginary} \odot u_{imaginary} \\ r_{real} \odot u_{imaginary} + r_{imaginary} \odot u_{real} \end{array} \right] \qquad (28)$$

The output key for each copy, $r_{o,s}$, is permuted by the same matrix as the copy's input key:

$$r_{o,s} = \left[\begin{array}{cc} P_s & 0 \\ 0 & P_s \end{array} \right] r_o \qquad (29)$$

Finally, the cells (memory trace) are read out by averaging the copies:

$$h_t = g_o \odot \text{bound}\left(\frac{1}{N_{copies}} \sum_{s=1}^{N_{copies}} r_{o,s} \circledast c_{s,t} \right) \qquad (30)$$

Note that permutation can be performed in $O(N_h)$ computations. Additionally, all copies can be updated in parallel by operating on tensors of size $N_{copies} \times N_h$.

On some tasks, we found that learning speed was improved by not feeding h_{t-1} to the update u: namely, W_{hu} is set to zero in Equation 18, which causes u to serve as an embedding of x_t. This modification was made for the episodic copy, XML modeling, and variable assignment tasks below.

6. Experiments

All experiments used the Adam optimisation algorithm (Kingma & Ba, 2014) with no gradient clipping. For experiments with synthetic data, we generate new data for each training minibatch, obviating the need for a separate test data set. Minibatches of size 2 were used in all tasks beside the Wikipedia task below, where the minibatch size was 10.

We compared Associative LSTM to multiple baselines:

LSTM. We use LSTM with forget gates and without peepholes (Gers et al., 2000).

Permutation RNN. Each sequence is encoded by using powers of a constant random permutation matrix as keys:

$$h_t = Ph_{t-1} + Wx_t \qquad (31)$$

Table 1. Network sizes on the episodic copy task.

Network	N_h	Relative speed	#parameters
LSTM	128	1	72,993
LSTM nH=512	512	0.18	1,078,305
Associative LSTM	128 (=64 complex numbers)	0.22 ($N_{copies} = 1$) 0.16 ($N_{copies} = 4$) 0.12 ($N_{copies} = 8$)	65,505
Permutation RNN	256	2.05	5,642
Unitary RNN	256 (=128 complex numbers)	0.24	6,666
Multiplicative uRNN	256	0.23	10,506

Only the input and output weights are learned. Representing sequences by "permuting sums" is described in Kanerva (2009).

Unitary RNN. Arjovsky et al. (2015) recently introduced recurrent neural networks with unitary weight matrices.[1] They consider dynamics of the form

$$h_t = f(W h_{t-1} + V x_t) \tag{32}$$

where W is a unitary matrix ($W^\dagger W = I$). The product of unitary matrices is a unitary matrix, so W can be parameterised as the product of simpler unitary matrices. In particular,

$$h_t = f(D_3 R_2 F^{-1} D_2 P R_1 F D_1 h_{t-1} + V x_t) \tag{33}$$

where D_3, D_2, D_1 are learned diagonal complex matrices, and R_2, R_1 are learned reflection matrices. Matrices F and F^{-1} are the discrete Fourier transformation and its inverse. P is any constant random permutation. The activation function $f(h)$ applies a rectified linear unit with a learned bias to the modulus of each complex number. Only the diagonal and reflection matrices, D and R, are learned, so Unitary RNNs have fewer parameters than LSTM with comparable numbers of hidden units.

Multiplicative Unitary RNN. To obtain a stronger baseline, we enhanced the Unitary RNNs with multiplicative interactions (Sutskever et al., 2011) by conditioning all complex diagonal matrices on the input x_t:

$$\hat{r} = W_{xr} x_t \tag{34}$$

$$D = \begin{bmatrix} \text{diag}(\cos(\hat{r})) & -\text{diag}(\sin(\hat{r})) \\ \text{diag}(\sin(\hat{r})) & \text{diag}(\cos(\hat{r})) \end{bmatrix} \tag{35}$$

6.1. Episodic Tasks

6.1.1. EPISODIC COPY

The copy task is a simple benchmark that tests the ability of the architectures to store a sequence of random characters and repeat the sequence after a time lag. Each input

[1]We were excited to see that other researchers are also studying the benefits of complex-valued recurrent networks.
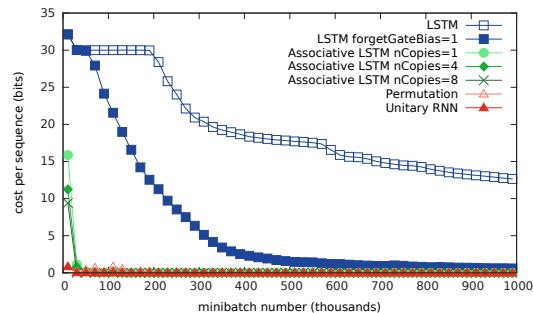

Figure 3. Training cost per sequence on the fixed-length episodic copy task. LSTM learns faster if the forget gate bias is set to 1. Associative LSTM was able to solve the task quickly without biasing the forget gate.

sequence is composed of 10 random characters, followed by 100 blanks, and a delimiter symbol. After the delimiter symbol is presented, networks must reproduce the first 10 characters, matching the task description in Arjovsky et al. (2015). Although copy is not interesting *per se*, failure on copy indicates an extreme limitation of a system's capacity to memorise.

Arjovsky et al. (2015) presented very good results on the copy task using Unitary RNNs. We wanted to determine whether Associative LSTM can learn the task with a similarly small number of data samples. The results are displayed in Figure 3. The Permutation RNN and Unitary RNN solve the task quickly. Associative LSTM solved the task a little bit slower, but still much faster than LSTM. All considered, this task requires the network to store only a small number of symbols; consequently, adding redundancy to the Associative LSTM, though not harmful, did not bestow any benefits.

We considered this variant of the copy task too easy since it posed no difficulty to the very simple Permutation RNN. The Permutation RNN can find a solution by building a hash of the input sequence (a sum of many permuted inputs). The output weights then only need to learn to classify the hash codes. A more powerful Permutation RNN
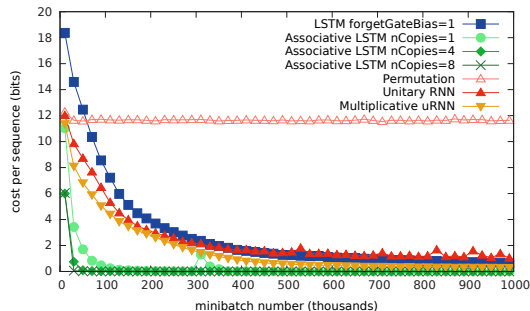
Figure 4. Training cost per sequence on the episodic copy task with variable-length sequences (1 to 10 characters). Associative LSTM learns quickly and almost as fast as in the fixed-length episodic copy. Unitary RNN converges slowly relative to the fixed-length task.

could use a deep network at the output.

To present a more complex challenge, we constructed one other variant of the task in which the number of random characters in the sequence is not fixed at 10 but is itself a variable drawn uniformly from 1 to 10. Surprisingly, this minor variation compromised the performance of the Unitary RNN, while the Associative LSTM still solved the task quickly. We display these results in Figure 4. We suspect that the Permutation RNN and Unitary RNN would improve on the task if they were outfitted with a mechanism to control the speed of the dynamics: for example, one could define a "pause gate" whose activation freezes the hidden state of the system after the first 10 symbols, including possible blanks. This would render the variable-length task exactly equivalent to the original.

Table 1 shows the number of parameters for each network. Associative LSTM has fewer parameters than LSTM if the W_{hu} matrix in Equation 18 is set to zero and the gates are duplicated for the real and the imaginary parts. Additionally, the number of parameters in Associative LSTM is not affected by the number of copies used; the permutation matrices do not add parameters since they are randomly initialised and left unchanged.

6.2. Online Tasks

As important as remembering is forgetting. The following tasks consist of continuous streams of characters, and the networks must discover opportune moments to reset their internal states.

6.2.1. XML MODELING

The XML modeling task was defined in Jozefowicz et al. (2015). The XML language consists of nested (context-free) tags of the form "`<tag1><tag2>`
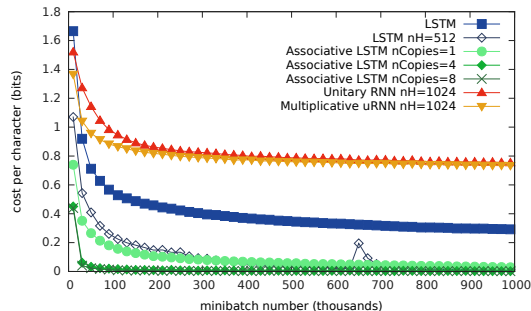

Figure 5. Training cost on the XML task, including Unitary RNNs.
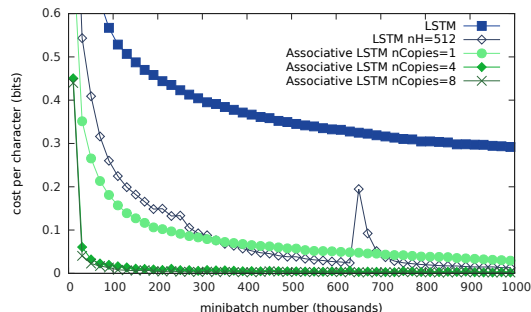

Figure 6. Training cost on the XML task. LSTM and Associative LSTM with 128 hidden units are also compared to a larger LSTM with 512 units.

`...</tag2></tag1>`". The input is a sequence of tags with names of 1 to 10 random lowercase characters. The tag name is only predictable when it is closed by "`</tag>`", so the prediction cost is confined to the closing tags in the sequence. Each symbol must be predicted one time step before it is presented. An example sequence looks like:

`<xkw><svgquspn><oqrwxsln></oqrwxsln></svgquspn><jrcfcacaa></jrcfcacaa></xk...`

with the cost measured only on the underlined segments. The XML was limited to a maximum depth of 4 nested tags to prevent extremely long sequences of opened tags when generating data. All models were trained with truncated backpropagation-through-time (Williams & Peng, 1990) on windows of 100 symbols.

Unitary RNNs did not perform well on this task, even after increasing the number of hidden units to 1024, as shown in Figure 5. In general, to remain competitive for online tasks, we believe Unitary RNNs need forget gates. For the rest of the experiments in this section, we excluded the Unitary RNNs since their high learning curves skewed the plots.

The remaining XML learning curves are shown in Figure 6. Associative LSTM demonstrates a significant advan-
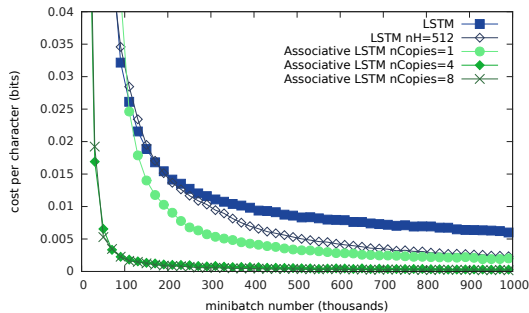
Figure 7. Training cost on the variable assignment task.

Figure 8. Training cost on the arithmetic task.

Figure 9. Training cost on the arithmetic task when using Associative LSTM with 3 writing and reading heads.

tage that increases with the number of redundant copies. We also added a comparison to a larger LSTM network with 512 memory cells. This has the same number of cells as the Associative LSTM with 4 copies of 128 units, yet Associative LSTM with 4 copies still learned significantly faster. Furthermore, Associative LSTM with 1 copy of 128 units greatly outperformed LSTM with 128 units, which appeared to be unable to store enough characters in memory.

We hypothesise that Associative LSTM succeeds at this task by using associative lookup to implement multiples queues to hold tag names from different nesting levels. It is interesting that 4 copies were enough to provide a dramatic improvement, even though the task may require up to 40 characters to be stored (e.g., when nesting 4 tags with 10 characters in each tag name).

6.2.2. VARIABLE ASSIGNMENT

The variable assignment task was designed to test the network's ability to perform key-value retrieval. A simple synthetic language was employed, consisting of sequences of 1 to 4 assignments of the form *s(variable,value)*, meaning "set variable to value", followed by a query token of the form *q(variable)*, and then the assigned value the network must predict. An example sequence is the following (prediction targets underlined):

```
s(ml,a),s(qc,n),q(ml)a.s(ksxm,n),s(u,v)
,s(ikl,c),s(ol,n),q(ikl)c.s(...
```

The sequences are presented one character at a time to the networks. The variable names were random strings of 1 to 4 characters, while each value was a single random character.

As shown in Figure 7, the task is easily solved by Associative LSTM with 4 or 8 copies, while LSTM with 512 units solves it more slowly, and LSTM with 128 units is again worse than Associative LSTM with a single copy. Clearly, this is a task where associative recall is beneficial, given Associative LSTM's ability to implement associative
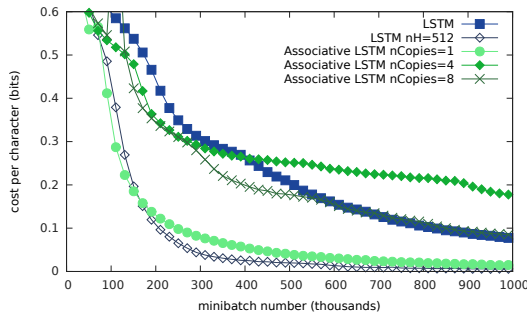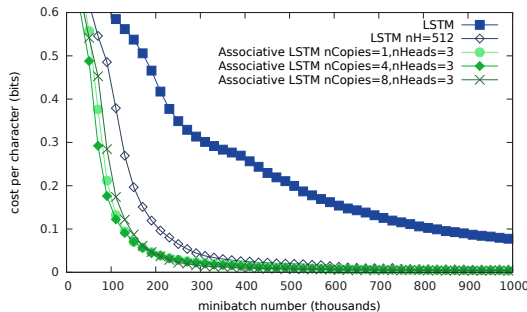
arrays.

6.2.3. ARITHMETIC

We also evaluated Associative LSTM on an arithmetic task. The arithmetic task requires the network to add or subtract two long numbers represented in the input by digit sequences. A similar task was used by Jozefowicz et al. (2015).

An example sequence is:

```
-4-98308856=06880389-]-981+1721=047]-10
+75723=31757]8824413+...
```

The character "]" delimits the targets from a continued sequence of new calculations. Note that each target, "−98308860, . . .", is written in reversed order: "06880389−". This trick, also used by Joulin & Mikolov (2015) for a binary addition task, enables the networks to produce the answer starting from the least significant digits. The length of each number is sampled uniformly from 1 to 8 digits.

We allowed the Associative LSTM to learn the W_{hu} matrix in Equation 18, which enables the cell state to compute a carry recurrently. Network parameters and execution speeds are shown in Table 2.

Table 2. Network sizes for the arithmetic task.

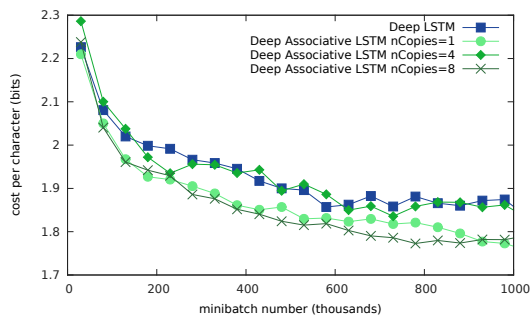| Network	N_h	Relative speed	#parameters
LSTM	128	1	$117,747$
LSTM nH=512	512	0.15	$1,256,691$
Associative LSTM	128 (=64 complex numbers)	0.19 ($N_{copies} = 1$) 0.15 ($N_{copies} = 4$) 0.11 ($N_{copies} = 8$)	$131,123$
Associative LSTM nHeads=3	128 (=64 complex numbers)	0.10 ($N_{copies} = 1$) 0.08 ($N_{copies} = 4$) 0.07 ($N_{copies} = 8$)	$775,731$

Figure 10. Test cost when modeling English Wikipedia.

Associative LSTM with multiple copies did not perform well on this task, though the single-copy variant did well (Figure 8). There is a subtle reason why this may have occurred. Associative LSTM is designed to read from memory only one value at a time, whereas the addition task requires retrieval of three arguments: two input digits and a carry digit. When there is only one copy, Associative LSTM can learn to write the three arguments to different positions of the hidden state vector. But when there are multiple copies, the permutation matrices exclude this solution. This is not an ideal result as it requires trading off capacity with flexibility, but we can easily construct a general solution by giving Associative LSTM the ability to read using multiple keys in parallel. Thus, we built an Associative LSTM with the ability to write and read multiple items in one time step simply by producing N_{heads} input and output keys.

As shown in Figure 9, adding multiple heads helped as Associative LSTM with 3 heads and multiple copies consistently solved the task.

6.2.4. WIKIPEDIA

The last task is the sequence prediction of English Wikipedia (Hutter, 2006), which we used to test whether Associative LSTM is suitable for a natural language processing task.

The English Wikipedia dataset has 100 million characters of which we used the last 4 million as a test set. We used larger models on this task but, to reduce training time, did not use extremely large models since our primary motivation was to compare the architectures. LSTM and Associative LSTM were constructed from a stack of 3 recurrent layers with 256 units by sending the output of each layer into the input of the next layer as in Graves et al. (2013).

Associative LSTM performed comparably to LSTM (Figure 10). We expected that Associative LSTM would perform at least as well as LSTM; if the input key r_i is set to the all 1-s vector, then the update in Equation 27 becomes

$$c_{s,t} = g_f \odot c_{s,t-1} + g_i \odot u \qquad (36)$$

which exactly reproduces the cell update for a conventional LSTM. Thus, Associative LSTM is at least as general as LSTM.

7. Why use complex numbers?

We used complex-valued vectors as the keys. Alternatively, the key could be represented by a matrix A, and the complex multiplication then replaced with matrix multiplication Ax. To retrieve a value associated with this key, the trace would be premultiplied by A^{-1}. Although possible, this is slow in general and potentially numerically unstable.

8. Conclusion

Redundant associative memory can serve as a new neural network building block. Incorporating the redundant associative memory into a recurrent architecture with multiple read-write heads provides flexible associative storage and retrieval, high capacity, and parallel memory access. Notably, the capacity of Associative LSTM is larger than that of LSTM without introducing larger weight matrices, and the update equations of Associative LSTM can exactly emulate LSTM, indicating that it is a more general architecture, and therefore usable wherever LSTM is.

Acknowledgments

References

Arjovsky, Martin, Shah, Amar, and Bengio, Yoshua. Unitary evolution recurrent neural networks. *arXiv preprint arXiv:1511.06464*, 2015.

Gers, Felix A, Schmidhuber, Jürgen, and Cummins, Fred. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

Graves, Alan, Mohamed, Abdel-rahman, and Hinton, Geoffrey. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6645–6649. IEEE, 2013.

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Grefenstette, Edward, Hermann, Karl Moritz, Suleyman, Mustafa, and Blunsom, Phil. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pp. 1819–1827, 2015.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hutter, Marcus. The human knowledge compression contest. 2006. URL http://prize.hutter1.net.

Joulin, Armand and Mikolov, Tomas. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28*, pp. 190–198. 2015.

Jozefowicz, Rafal, Zaremba, Wojciech, and Sutskever, Ilya. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 2342–2350, 2015.

Kanerva, Pentti. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive Computation*, 1(2):139–159, 2009.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Plate, Tony. Holographic reduced representation: Distributed representation for cognitive structures. 2003.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Sukhbaatar, Sainbayar, Weston, Jason, Fergus, Rob, et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, pp. 2431–2439, 2015.

Sutskever, Ilya, Martens, James, and Hinton, Geoffrey E. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017–1024, 2011.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc VV. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

Williams, Ronald J and Peng, Jing. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

Zaremba, Wojciech and Sutskever, Ilya. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.